Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M.L. Kersten, A.P.J.M. Siebes, C.A. van den Berg

Using a graph rewriting system for databases

Computer Science/Department of Algorithmics & Architecture Report CS-R8805 January

Using a Graph Rewriting System for Databases

M.L. Kersten
A. Siebes
C. van den Berg

*Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*

ABSTRACT

A major portion of a database management system deals with maintaining graph-like structures using a variety of data structures and algorithms. In this paper we describe a method to partly automate their construction using a rewrite system for colored graphs. Our approach provides both a formal basis for the description of the DBMS semantics and it will improve the exploitation of the potential parallelism in a DBMS.

1986 CR Categories: D.2.1 [**Software Engineering**]: Requirements/Specification - languages D.3.2 [**Programming Languages**]: Language classification - nonprocedural very high-level languages, D.3.3 [**Programming Languages**]: Language constructs - concurrent programming and control structures H.2.0 [**Database Management**]: Physical design - access methods.

1980 Mathematics Subject Classification: 69D21, 69D42, 69D43, 69H22.

Key words & phrases: specifying database structures.

1. Introduction

Most programs can be conveniently classified as input-output transformers. They accept commands and generate output after a series of transformations. In general, the program state is forgotten afterwards and special actions are needed to retain portions of the data structures for future invocations of the same (or another) program.

A prototypical example of such as system is a language parser. Their development is greatly simplified by the compiler generators currently available. They take a complete problem specification and generate a source program that handles the transformation. Alternatively, they generate an encoding of the specification such that a small interpreter can simulate the intended behavior. Moreover, the large body of theory on grammars and parsing techniques makes them verifyable and efficient.

The state of the art in the design of database management systems is comparable with the pre-compiler generator era. Besides the parser generators applied for input analysis, few automated methods exist that aid in their construction. At best, part of the programming work can be avoided by using programming libraries with the required functionality. But still an investment of many man-years is required to construct a new DBMS.

Several research groups are actively searching for techniques that alleviate the problems incurred by the software complexity of a DBMS. A few of them are mentioned here for

Report CS-R8805
Centre fof Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

reference.

The computer science group at the University of Connecticut is working on a compiler that produces object-oriented database systems [Maryanski86]. Experience to date has resulted in the generation of prototype systems for business processing, mechanical CAD, and office automation. As an aside they have developed a methodology to aid the user in specifying the properties of the DBMS.

At the IBM Almaden Research Center the Starburst project [Schwarz86] investigates extensibility of traditional relational systems for new applications and technologies. The focus on parameterization of external data storage, storage management, access methods, abstract data types and complex objects. They assume that extensibility is provided by a knowledgeable programmer.

The GENESIS project [Batory86] at the University of Texas, Austin, aims at the development of an extensible database management system. Their prime focus is extensibility of the storage and access path modules. An interesting aspect of this project is their use of a formal model to describe the logical and physical database organization. This model provides the handle to partly automate the system generation.

Unlike the previous projects we use a simple formal model for the specification of the DBMS structure and its operational semantics. In our approach the DBMS is described as a rewriting system for colored graphs. The reasons for this can be summarized as follows.

- The internal organization is naturally mapped onto a directed graph. In fact, most of the DBMSs' speed is obtained from chosing the proper graph-like data structure.
- A graph rewrite system can be rigorously formalised. Thereby, it aids in the formal verification of the final DBMS.
- A graph rewrite system supports declarative programming, i.e. rewrites are applied independent of their ordering within their specification and the state graph. It is up to the rule interpreter to locate the rules and subgraphs for rewriting.
- If the order in which graph rewrite actions are applied does not effect the final outcome and converges to a normal form then the rewrite set is said to satisfy Church-Rosser. This property provides the basis for further optimization and parallel implementations.
- The specification is expected to be much smaller than the code generated for the DBMS. In combination with the tools for the formal analysis it leads to a system that is easier to maintain.

Our work can be considered an outgrowth of the work of Weber [Weber78]. He gives a formal description of a graph rewrite system and shows how the CODASYL database structures can be formally described as a set of rewrite rules. We also focus on maintaining a large search space structured as a directed graph and intend to use the rewrite rules to derive search algorithms automatically.

Related work on graph rewriting systems for database management has also been done by Furtado et. al [Furtado78]. and Ehrig et. al [Ehrig]. Both papers describe how an information system based on a graph rewrite system can be modelled. The latter also deals with synchronization of parallel execution of the actions. However, as far as we are aware, no attempt has been made to generalize the approach to specify and construct a general database management system.

The rest of this paper is organized as follows. In Section 2 we present database structures and show how they can be specified in a graph rewriting system (GRS). Section 3 introduces the pattern matching and rewrite strategy. In section 4 we give a more complicated example and indicate how other components of a DBMS can be formulated within GRS. We conclude with a summary and indication of future research.

2. Specifying database structures in GRS

In this section we show how a rudimentary database system can be cast into as a graph rewriting problem. The presentation is also meant to introduce the concepts of our graph rewriting system and its specification language, called GRS, in an informal and intuitive manner. The formal definition of our rewrite system is beyond the scope of this paper.

2.1. The State Graph of a Single Relation

Through drastic simplification, a DBMS can be viewed as an input-output system in which the database is structured as a directed graph. The nodes in this graph are <color,value> pairs that will represent relations, tuples, attributes, etc.. The color is an identifier by which a node can be locally identified.

For example, consider a database with a single unary relation of integers. Moreover, assume that the tuples are organized internally as an ordered list. Then a state of the database can be described with the graph shown in Fig 1.

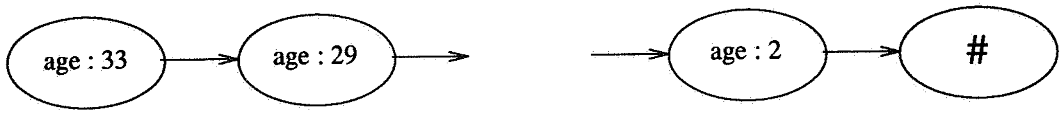

Figure 1 The state graph of a unary relation

The node colored # is called a crystallization point. It is used to distinguish nodes part of the database representation from garbage. When the rewrite system is quiescent, i.e. no rewrite rule can be applied any more, then all nodes should be connected (indirectly) with precisely one crystallization point. Those that don't are considered garbage and are removed.

In this paper we mostly use a linear representation of the graphs. Fig. 2 introduces and illustrates our notational conventions. Names starting with a lower case letter represent colors, those with an upper case are variables. Each variable in the pattern will refer to a single node in the graph.

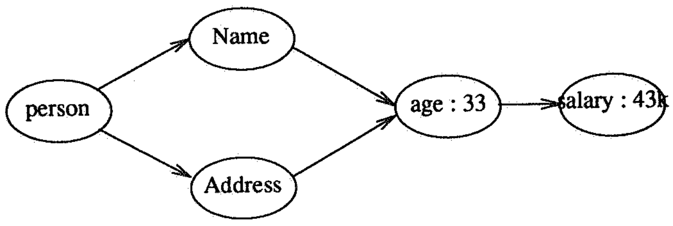

Figure 2 person→[Name,Address]→age:33→salary:43k

2.2. The Rewrite Rules for Maintaining the Unary Relation

The rudimentary database is augmented with three operations: *insert*, *delete*, and *print*. The *insert* accepts a new tuple from input and inserts it at the proper place in the data structure. Duplicate tuples are not accepted. The *delete* command removes a tuple from the database, provided it has been stored previously. The *print* statement copies all tuples to output, its discussion, however, is postponed to the next section.

The relation is manipulated through input from the user. Proper input messages are accepted by the IO interface and translated into a request for a graph rewrite action. At some point this request will be picked up by the graph rewriter and matched against the state graph. If it happens to find a proper subgraph it will make the modifications requested, possibly by generating other rewrite requests. If no match can be made then the rewrite request is considered a no-op and removed from further consideration.

The *insert* operation specifies *how* or *where* the new tuple should be stored in the state graph. The former leads to a procedural specification, which, for example, describes how one searches the state graph from a crystallization point to the subgraph of interest. The latter approach, called a declarative specification, describes the properties of the subgraph of interest and leaves the actual search to the system. A declarative specification is often preferable, because it hides implementation details as much as possible. A declarative description of an *insert* rule is given below using the GRS notation (See appendix).

rule	insert(V)
pattern	R→S
with	R=S=V=age
where	R>V>S
becomes	R→V→S

Figure 3 The *insert* operator

This text fragment is interpreted as follows. The **rule** clause describes the name of the rewrite request and its arguments. The arguments are (complex) graph-like patterns which should match against the pattern arguments submitted during a rewrite requests. In our example, a valid request is *insert*(*age*:31). An invalid request would be *insert*(*employee*→[*age*:32,*name*:'john']).

The **pattern** specifies the structure of the subgraph in which the rewrite action should have its effect. The rules for matching are described shortly in more detail. In our example, it selects a subgraph of two directly linked nodes R and S.

The **with** and **where** parts further limit the potential subgraphs to those that satisfy a coloring and value constraint, respectively. The **with** keyword is followed by a boolean expression involving the node colors only. It is used to check a node for a specific color (R=age) or compares node colors (R=S). Similarly, the **where** is followed by an expression involving the value parts of the nodes (R>V).†

The last clause, **becomes**, describes how the graph should be redrawn. In doing so, it assumes that all arcs participating in the pattern match are already removed. In some cases this 'damage' should be undone explicitly. During rewrite new nodes can be constructed using constant values and color names (*age*:23), and through dereference of variables (R:S). The system enforces a single-assignment rule, i.e. the only way to change the value (or color) of a node is by creating a new node with the desired properties. In our example, it suffices to rewrite the graph to a new sublist with the element in place.

Unfortunately, the rule in Fig. 3 alone is not sufficient to maintain the unary relation depicted in Fig. 1, because it does not deal with the boundary conditions: the list is empty ([]->#), the new element is greater then all stored elements, and the new element is smaller then all stored elements. The following fragments complete its specification. They illustrate the use of the crystallization point, checking for omission of arcs into the node H ([]→H), and syntactic shorthands (age→#). ††

rule	insert(age)	**rule**	insert(V)	**rule**	insert(age)
pattern	[]→#	**pattern**	H→#	**pattern**	[]→H
becomes	age→#	**with**	H=V=age	**with**	H=age
		where	H>V	**where**	V>H
		becomes	H→V→#	**becomes**	V→H

Figure 4 Boundary conditions for *insert*

† If an expression leads to an error (typing, arithmetic error) then the rule invocation is considered erroneous and terminated. Changes already made to the state graph are undone.
†† The construct []->A means in(A)=0.

The *delete* operator is less complicated, because all nodes, except the header and tail node, have two neighbors. Therefore, two rewrite rules suffice to describe its semantics (Fig 5). The first rule deals with the general case. The second rule takes care of the boundary condition. It does not need a **becomes** part, because all arcs used in the **pattern** match are automatically removed.

rule	delete(V)	**rule**	delete(V)
pattern	R→S→T	**pattern**	[]→S→T
with	S=V	**with**	S=V
where	S=V	**where**	S=V
becomes	R→T		

Figure 5 The *delete* operation

2.3. The Input-Output Interfaces

Since we intend to use the graph rewrite system for programming purposes we need input/output interfaces. To simplify matters, we assume that the user behind the terminal communicates graph-like structures, much the same as our linear notation. Then, conceptually, each input message can be interpreted as a rewrite request.

To avoid interference and to maintain global, application dependent invariants we will describe the acceptable input messages explicitly. The form and interpretation of the **input** interface is similar to the rewrite rules discussed above. It differs by accepting messages and generating rewrite requests. It may, however, also inspect and alter the state graph directly. The **output** interface is handled similarly. Rewrite requests of interest are located and transformed into messages. Two example interface specifications are shown in Fig. 6. Notice, that the rewrite actions in *print_age* are meant for the display handler only which determines the final display format.

input	insert_age(V)	**output**	print_age(A)
with	V=age	**with**	A=age
where	V>=0	**becomes**	'The age is ' A
becomes	insert(V)		

Fig. 6 Two interface rules.

3. Graph rewriting

In this section we introduce the policy for pattern matching and rewrite strategy.

3.1. Pattern matching

In our explanation of rewrite rules we skipped the details of matching the pattern against a portion of the state graph. This, however, can be done in several ways. For our graph rewrite system we have chosen the following policy.

a) The pattern describes a connected graph.

The reasons are that all rules involve an explicit, limited subgraph and that we do not have to consider the side-effects of our rewrites on the otherwise unspecified paths between two graph components. In a sense, it is a requirement for specification precision. It states that sufficient context is made explicit such that all the effects can be derived from the description only.

b) A name in the pattern is associated with at most one node in the state graph.

The names in the pattern are node variables. Each name is related with a single node in the state graph during pattern matching. Moreover, this criterion ensures that nodes in the state graph play a single role during a rewrite. As a result the number of nodes involved in any pattern and their behavior can be determined statically.

c) A node match succeeds iff all or none of its incoming (outgoing) arcs are used in the pattern.

To illustrate this criterion consider the pattern A→B→C. Let *in*(X) and *out*(X) denote the number of incoming and outgoing arcs of X, respectively. Then a match with a subgraph succeeds if the following expression holds:

out(A)=1 **and** *in*(B)=1 **and** *out*(B)=1 **and** *in*(C)=1

Thus, we know that no more than one path emanates from A, no other path passes through B, and no other path ends in C. The incoming arcs of A (outgoing from C) are of no interest. They are also called boundary nodes. †

The reason for introducing this criterion is that all path changes are now made explicit. In particular, cycles in the graph can be detected merely by inspecting individual rule descriptions. This also makes life of the GRS programmer easier, because cycles do not result of side-effects.

The drawback is twofold. Sometimes more context should be specified than really necessary for the intended rewrite (Section 4.1). Moreover, some graphs can not be dealt with any more. For example, it prohibits a state graph that represents a zig-zag data structure of arbitrary size (Fig. 7).

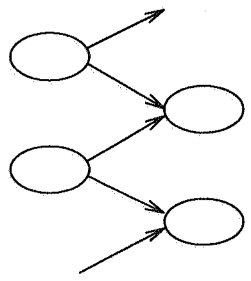
Figure 7 A zig-zag data structure

3.2. The rewrite strategy

The second important issue, called the rewrite strategy, is when and how a rule is selected. In general, a rule may be applicable repeatedly and various rules may be applicable at the same time. A fair policy is to pick a rule at random and select an arbitrary subgraph that will match a portion of the state graph. Following the color and value expressions are evaluated. If either does not hold then the rule invocation is cancelled. Otherwise we have found a matching subgraph, all arcs specified in the pattern are removed, and the rewrite actions are interpreted.

The outcome of the rewrite action is a subgraph that should be embedded into the state graph. To keep the description of our rules simple we have chosen the following policy.

d) The new subgraph is embedded in the new state through the boundary nodes.

Another policy issue is whether constraints should be imposed on the new subgraph similar to the pattern matching. Again we have chosen to make changes explicit through their rules. This translates to the following constraint.

e) The new subgraph patterns can be matched against the resulting state graph.

To illustrate both issues, reconsider the pattern A→B→C. Then embedding is limited to either A and C (or both). Moreover, the rewrite actions are not allowed to create new incoming (outgoing) arcs into A (from C). This means that the boundary nodes of the pattern are also

† A→[] means out(A)=0

boundary nodes of the new subgraph.

This rewrite policy can be used to simulate more complex policies using the state graph as a black board on which the control information is written. However, a recurring situation is that a rewrite action should be applied once to all subgraphs in the state graph that satisfy the pattern. This case can better be indicated by the programmer explicitly, because it leads to unanticipated side-effects. Moreover, it complicates the envisioned parallel execution. The *print* operator illustrates this.

The purpose of the *print* operator is to copy all tuples to output. This task can be described with the rules shown in Fig. 8. A sample rewrite sequence is shown in Fig. 9. The rule at the left accepts the *print* command from the user and matches it with the head of the tuple list. Following, the rule at the right copies each tuple and advances the marker *nextprint* in the state graph. It also shows activation of a rewrite by the state.

rule	print	**pattern**	H→[nextprint,T→T2]	**pattern**	H→[nextprint,#]
pattern	[]→H→T	**becomes**	H→T→[nextprint,T2]	**becomes**	H→#
becomes	H→[nextprint,T]		print_age(H)		print_age(H)

Figure 8 The *print* operator

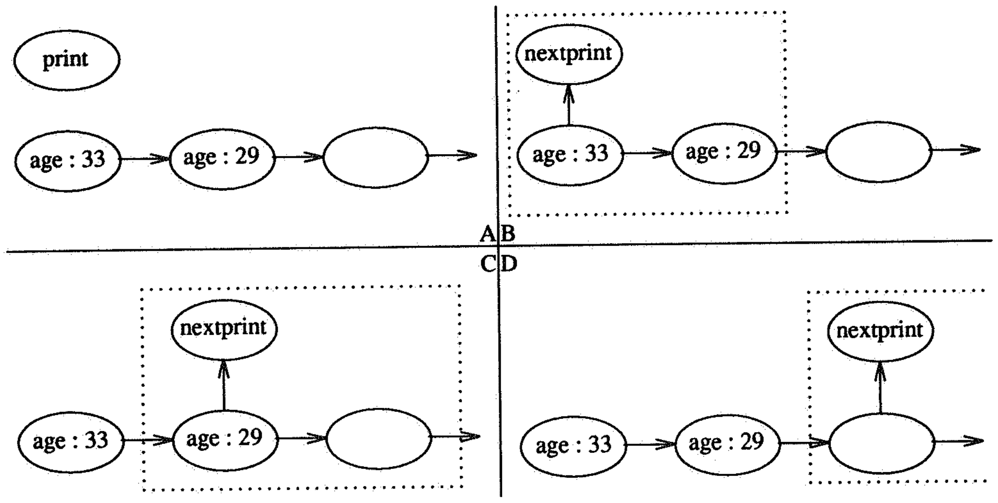

Figure 9 A sample rewrite sequence

This specification of the print command uses a procedural interpretation of the rewrite process, because it incorporates flow-of-control information in the state graph. The effect is that there is at most one possible rewrite. Moreover, it is likely to block other rewrite requests interested in the node to be printed, because it has one an additional arc. A more concise and declarative specification reads as follows.

pattern all	age
becomes	print_age(age)

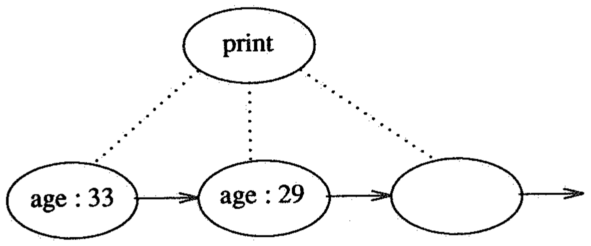

Figure 10 The declarative *print* operator

The keyword **all** indicates that this rule should be applied for all possible tuples in the state, before another rule is selected. The **all** indicator is allowed if any two selected subgraphs overlap at most on the boundary nodes and this property can be established without concern about the actual state graph. Unrestricted use would lead to various interpretation problems. Either we could first select all patterns and then apply the rewrite collectively, or we could iterate over the rule as long as we can find a new subgraph. The former easily leads to situations where the effect of the rewrite differ, due to overlap. The latter may lead to an infinite rewrite sequence. Examples of both cases are shown in Fig. 11.

pattern	H→R→S	**pattern**	H→R
with	H=age **and** R=age	**with**	H=age
becomes	H->[tag,R→[tag,S]]	**becomes**	H→age:4→R

Figure 11 Two anomalies

3.3. Properties of rewrite sets

Since the rewrite policy picks rules and subgraphs at random a naive rule set may lead to outcomes that depend on the order of rule selection. To avoid such non-deterministic behavior the rules involved should be *confluent* and *strongly* normalised, called Church-Rosser. The former intuitively means that if a rule triggers two rewrite requests the final outcome will not depend on which is chosen as immediate successor. The latter means that rewrite sequences will terminate. Our prototype interpreter assumes that the rule set is non-ambiguous and Church-Rosser.

Unfortunately, there does not exists a general decision procedure that determines confluency and strong normalisation. However, algorithms exist that aid in the construction of proper rule sets. They are envisioned to be part of the programming environment for GRS.

4. The DBMS internals

In this section we sketch how our rewrite system can be used to specify the behavior of portions of a DBMS. The first topic addressed is a binary search tree, which is normally used internally to improve the DBMS performance. Following, we indicate how a relation scheme can be transformed into a set of rewrite rules and how such a schema is extended. Last, but not least, we indicate how queries are handled.

In this expository we assume that an interpreter for GRS exists that maintains the state graph using stable storage. Conceptually, the user should supply both a command and the rewrite set that deals with it. New commands are only accepted when the interpreter is in a quiescent state, i.e. there is no applicable rewrite rule. The net effect of this simplification is that at this stage we do not have to introduce functions, meta-rules, and modularization primitives.

4.1. Data structures

Consider the case that we need a (non-balanced) binary search tree of integers. Moreover, there is only one such tree. A first partition of the problem is to distinguish between insertion of the first node and the n-th node. The former is also responsible for initialization of the tree structure. The rule and its effect for this case are shown in Fig. 12 and 13.a. The two nodes colored *end* are introduced to simplify subsequent insertion of nodes, because then each *node* has two integer valued sons.

rule	insert(node)
pattern	#→[]
becomes	#→node→[end:-∞, end:+∞]

Figure 12 Initialization of the binary tree

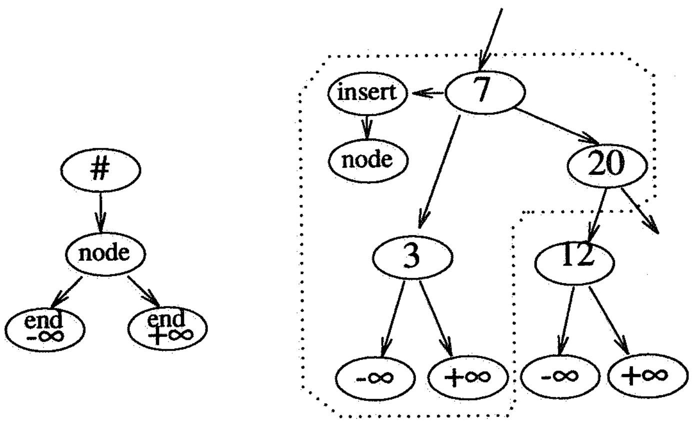

Figure 13 a) The initial state b) An portion of the binary tree

Handling the n-th case can specified in two ways: using a procedural rewrite or a declarative rewrite. The procedural rewrite locates the root node first and then descends the structure to locate an *end* point where the new node can be stored. The corresponding rules are shown in Fig. 14. Note that our matching strategy requires the description of a substantial context (indicated as a box in Fig. 13.b).

	insert son of root
rule	insert(N)
pattern	#→Root→[Son1,Son2]
becomes	#→Root→[Son1,Son2,insert→N]
	insert at leaf node
pattern	T→[Son1,Son2,insert→N]
with	Son1=end
where	(T>N>=Son1) **or** (T<=N<Son1)
becomes	T→[Son2, N→[end:-∞, end:+∞]]
	insert intermediate node
pattern	Root→[insert→N, Son1→[S1,S2], Son2]]
with	Root=node
where	(N<Root **and** Son1<Son2) **or** (Son1>=Son2 **and** N>=Root)
becomes	Root→[Son1→[insert→N,S1,S2], Son2]

Figure 14 A procedural rewrite policy

The declarative rewrite rule is also of interest, because it does not specify the exact sequence. Rather, it specifies the properties of an allowable location only. The associated rewrite rules are described below.

	create the tree structure
rule	createtree(node)
pattern	#
becomes	#→root:node → [node:- ∞ , node:+ ∞]

	insert a node		
rule	insert(node)	**rule**	insert(node)
pattern	#→R→[N,M]	**pattern**	#→R→[N,M]
with	N = end **and** R=node	**with**	M = node **and** R=node
where	N = -∞ **and** node < root	**where**	M = +∞ **and** node > root
becomes	R→[node→[end:-∞, end:+∞], M]	**becomes**	R→[node→[end:-∞, end:+∞],N]
rule	insert(node)	**rule**	insert(node)
pattern	A→[B→[C,D],E]	**pattern**	**A→[B→[C,D],E]**
with	**B=C=D=E=node**	**with**	**B=C=D=E=node**
where	**A>B>node>C**	**where**	**A>node and D>node>B**
becomes	**A→[B→[node→[C,end:+∞],D],E]**	**becomes**	**A→[B→[node→[D,end:-∞],C],E]**
rule	**insert(node)**	**rule**	**insert(node)**
pattern	**A→[B→[C,D],E]**	**pattern**	**A→[B→[C,D],E]**
with	**B=C=D=E=node**	**with**	**B=C=D=E=node**
where	**node>C and B>node>A**	**where**	**D>node>B>A**
becomes	**A→[B→[node→[C,end:+∞],D],E]**	**becomes**	**A→[B→[node→[D,end:-∞],C],E]**

Figure 15 A declarative rewrite policy

4.2. Tuple structures

A database contains more complex records than the integer fields considered in the preceding sections. In most exemplary relational databases we find an employee relation of the form *employee*[*name*,*address*,*city*,*age*]. Internally such a relation can be represented as a binary search tree of records.

In our approach we merely replicate the definition of the binary search tree of the preceding section, extending each node with three attributes. A portion of the new tree is shown in Fig. 18. In addition to the arcs we should extend the **with** and **where** part to cope with a compound comparison. The initialization rule is shown only.

rule	insert(employee→[name,address,city,age])
pattern	#→[]
becomes	employee→[name,address,city,age, end:-∞, end:+∞]

Figure 17 Initialization of the employee relation

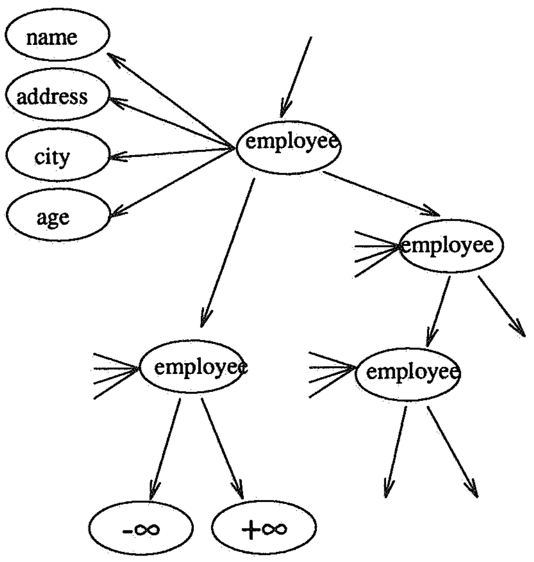

Figure 18 The representation of the employee relation.

4.3. Query handling

Querying a database can be seen as a modified print request, i.e. a selective part should be shown. Therefore, simple attribute queries can be directly transformed into a print rewrite rule extended with the appropriate **with** and **where** part.
For example, in Fig. 19 we show the rules for the queries "print all employees older than 32" and "show the name of employees", respectively. Notice that we undo the 'damage' introduced by the pattern match.

rule	query1
pattern all	employee→[name,address,city,age]
where	age>32
becomes	output(employee→[name,address,city,age])
	employee→[name,address,city,age]
rule	query2
pattern	employee→[name,address,city,age]
becomes	output(name)
	employee→[name,address,city,age]

Figure 19 Transformation of a query to a rule.

A drawback of this approach is that each query should first be transformed to a rewrite rule and included into the active rewrite set. We envision a more general query handling mechanism that accepts the rules directly from input can be designed when the rewrite set can be manipulated dynamically. This, however, is beyond the scope of this paper.

5. Summary and future research

In this paper we have shown that important components of a database management system, i.e. relation storage, insertion/deletion, and querying, can be formulated as a graph rewrite problem. For this we have introduced a graph rewriting specification language in which visibility of effects is highlighted. Moreover, the stringent format of the graph patterns supports static analysis which is deemed necessary for an efficient implementation.

The GRS language is devoid of flow-of-control primitives. It includes non-determinism to enable parallel evaluation and single-assignment rule to avoid side-effects. Both properties simplify reasoning about a set of rules and make a verifyable implementation of a GRS interpreter more attainable.

Two promising areas for future research should be pointed out. The prototype interpreter of our rewrite system used a brute force technique to locate a subgraph for rewriting. This technique can be improved significantly by automated constructing of a search algorithm for each rule. Such an algorithm should make efficient use of all information contained in the rule set.

Another area of interest is detection and exploitation of potential parallelism in the rewrite set. We hypothise to be in a good position, because each rule describes the smallest structure in which a single action makes sense and where the necessary access relationships are made explicit. This information need not be inferred from a program written in an imperative language. Following, the amount of parallelism is fully determined by the a number of non-overlapping applicable rewrite actions.

Acknowledgements

We thank F. Schippers for constructive comments on earlier versions of this paper.

References

[Batory86]
Batory, D.S., ''GENESIS: A Project to Develop an Extensible Database Management System,'' *Proceedings Int. Workshop on Object-Oriented Database Systems*, pp.206-207, Sep 1986.

[Ehrig]
Ehrig, H. and Kreowski, H.-J., ''Application of Graph Grammar Theory to Consistency, synchronization and scheduling of Data Base Systems,'' *Information Systems*, vol. 5, pp.225-238.

[Furtado78]
Furtado, A.L. and Veloso, P.A.S, ''Specification of Data Bases through rewriting rules,'' *Graph-Grammars and their Application to Computer Science*, pp.102-114, 1978.

[Maryanski86]
Maryanski, F., Bedell, J., Hoelscher, S., Hong, S., McDonald, L., Peckman, J., and Stock, D., ''The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems,'' *Proceedings Int. Workshop on Object-Oriented Database Systems*, pp.73-84, Sep 1986.

[Schwarz86]
Schwarz, P., Chang, W., Freytag, J.C., Lohman, G., McPherson, J., Mohan, C., and Pirahesh, H., ''Extensibility in the Starburst Database System,'' *Proceedings Int. Workshop on Object-Oriented Database Systems*, pp.85-92, Sep 1986.

[Weber78]
Weber, D., ''Transformation Programs for Data Graphs, a Tool for Specifying, Verifying and Implementing Data Types,'' pp. 246-263 in Graphs, Datastructures, Algorithms, ed. M. Nagl, H-J. Schneider, Springer-Verlag, Applied Computer Science 13 (1978).

Appendix A:The GRS specification language

Below we show the GRS grammar in BNF notation. The conventional meta symbols [] for optional and {} for repeating (>=0) constructs are used. The syntax for (boolean) expressions is left out for obvious reasons.

```
program         ::= {interface} {rule}.

interface       ::= input header body
                  | output header body.

header          ::= rulename '(' pattern {',' pattern } ')' .

rulename        ::= identifier .

rule            ::= [rule header ] pattern [all] pattern body.

body            ::= [with colorexpr ]
                    [where booleanexpr ]
                    [becomes rewrite {rewrite} ] .

pattern         ::= element {'→' pattern} .

element         ::= name | '[' pattern {',' pattern } ']'.

name            ::= variable | colorname | ''.

colorexpr       ::= name '=' name | name '~=' name | '(' colorexpr ')' | '~' colorexpr
                | colorexpr '&' colorexpr | colorexpr '|' colorexpr.

rewrite         ::= newelement {'→' rewrite} .

newelement      ::= name | name ':' expression | constant
                | '[' rewrite {',' rewrite} ']' .
```

The shorthands used in this paper are all textual. They do not affect the semantics of the language.

- A color name in a pattern is interpreted as a variable with a test on the actual color.

shorthand	expanded form		
pattern	red->X	**pattern**	Z→X
with	C	**with**	C **and** Z=red

- Rule arguments may be used within patterns as a shorthand for equality test.

shorthand	expanded form		
rule	p(X)	**rule**	p(X)
pattern	Z->X	**pattern**	Z→Y
with	C	**with**	C **and** Y=X
where	D	**where**	D **and** Y=X